DUT 2 Informatique – S3
M3103 – Algorithmique avancée

CM 06 – Collections et tables de hachage
Liens avec l'API Java

Équipe pédagogique : Camille Kurtz, Jacques Alès-Bianchetti, Simon Fuhlhaber

prénom.nom@parisdescartes.fr

18 août 2016

U S PC
Université Sorbonne Paris Cité

UNIVERSITÉ PARIS DESCARTES
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE

Plan

Plan

Introduction

Structures de données avancées

Nous étudierons dans ce support les structures de données suivantes :

- **collections** : listes, ensembles, itérateurs
- **structures associatives** : . . . tables de hachages

Nous nous attacherons également à apprendre à employer ces structures dans un langage de programmation classique. En **Java**, on trouve des classes pour ces structures dans le paquetage

`java.util`

(avec d'autres : pile (`Stack`), file (`Queue`), `Vector`, etc.)

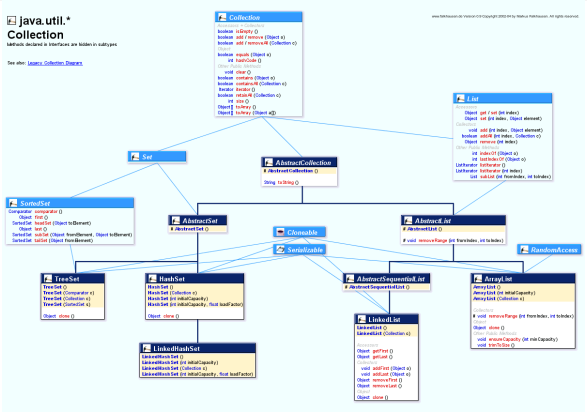
Manipulation de structures de données complexes où **des connaissances d'algorithmique avancée** deviennent indispensables

Plan

Premier regard sur les collections

Définition

- Une collection est un . .ensemble ordonné
- On trouve des collections de **comportements différents** (listes, ensembles, ...)
- **Une interface** `java.util.Collection<E>` **définit le contrat des collections**
- À partir de Java 1.5, les collections sont typées. `E` représente le type des éléments de la collection



Méthodes principales de `Collection<E>`

`boolean add(E e)` Ensures that this collection contains the specified element (optional operation)

`boolean contains(Object o)` Returns true if this collection contains the specified element, *i.e.*, $\exists e(o == null ? e == null : o.equals(e))$
$\rightarrow$ explique la signature de la méthode equals

`boolean isEmpty()` Returns true if this collection contains no elements

`Iterator<E> iterator()` Returns an iterator over the elements in this collection

`boolean remove(Object o)` Removes a single instance of the specified element from this collection, if it is present (optional operation)

`int size()` Returns the number of elements in this collection

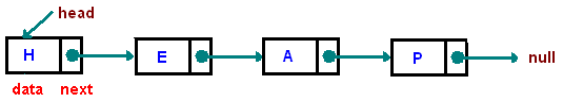
Plan

Retour sur les listes

Rappels

Une structure de données de type liste :

- est une suite d'éléments (suivant / précédent)
- est de taille non bordnée
- supporte les opérations : rechercher, supprimer, ajouter en tête (insérer après/avant un élément, en fin), etc.



List<E>

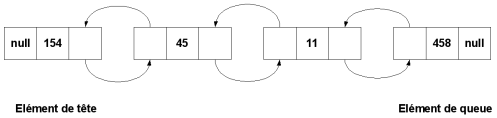
interface List<E> = collection ordonnée d'objets

2 classes :

- `ArrayList<E>` : listes implantées avec un tableau

  API Doc « The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires $\mathcal{O}(n)$ time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation. »

- `LinkedList<E>` : listes (doublement) chaînées



List<E> : méthodes complémentaires

Dans une liste les éléments sont ordonnés, la notion de position a un sens

`add(int index, E element)` ajout de l'élément à l'index-ième position

`E get(int index)` fournit l'index-ième élément de la liste.
IndexOutOfBoundsException - si (index < 0 || index >= size())

`boolean remove(int index)` supprime l'index-ième élément de la liste
(même exception)

`int indexOf(Object element)` indice de la première occurrence élément dans la liste, -1 si absent

`ListIterator<E>` itérateur pour listes doublement chaînées

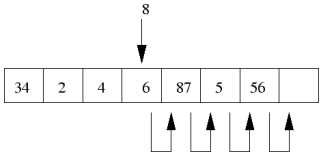
Complexité : listes implantées avec un tableau

Avantages

- L'accès au k-ième élément Theta(n)
- Peut permettre l'implantation de **fonctions** de recherche et de tri efficaces

Inconvénients

- **Le dépassement de capacité** peut être résolu en copiant le contenu dans un tableau plus grand, en $\Theta(n)$
- **La concaténation** est Theta(n)
- Nécessite de disposer d'un espace supplémentaire en $\Theta(n)$ pour ces deux dernières opérations



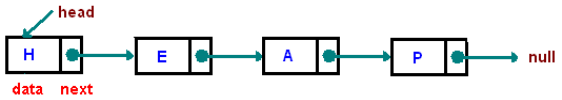
Complexité : listes chaînées modifiables

Avantages

- **Ajouter un élément en tête** : créer une nouvelle cellule, puis chaîner = mettre à jour le suivant en $\Theta(1)$

Inconvénients

- **Rechercher un élément** : ... Theta(n)
- **Suppression d'un élément** : recherche de la cellule à supprimer en se souvenant de la cellule précédente + suppression par dé-chaînage en $\mathcal{O}(n+1) = \Theta(n)$



Quoi utiliser ?

- `ArrayList` ssi ... intérations et accès indicé
- `LinkedList` ssi ... nombreuses insertions et suppression

Exemple de test de performances :

```
.../java/test$ java TestCollection2 20000 20000

*** insertion en tete LinkedList
20000 insertions ds LinkedList : 16 ms
*** insertion en tete ArrayList
20000 insertions dans ArrayList : 403 ms

*** remove LinkedList
20000 suppressions dans LinkedList : 8 ms
*** remove ArrayList
20000 suppressions dans ArrayList : 398 ms
```

Quoi utiliser ?

Stratégie

En cas de « non obligation » (ou de doute) sur le choix :
⇒ utiliser l'upcast vers l'interface associée à la collection pour faciliter le changement de choix d'implémentation

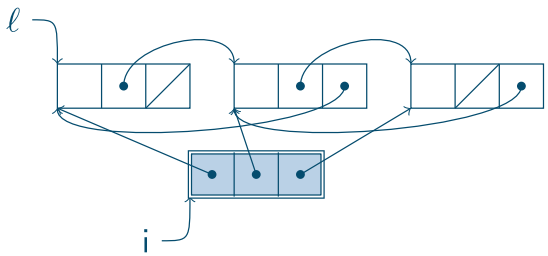
```
List<Livre> aList = new ArrayList<Livre>();
...
//traitements avec uniquement des methodes de
//l'interface List
...
//si besoin ulterieurement on peut changer en :
List<Livre> aList = new LinkedList<Livre>();
...
//memes traitements sans autre changement
```


Plan

Parcours d'une structure de données : itérateurs

Définition

- Un **itérateur** est une ... structure de données permettant le parcours
- Opérations supportées :
  - avancer, reculer
  - est_en_fin, est_en_début
  - valeur
  - insérer_apres, insérer_avant, supprimer



Exemple d'un itérateur employé pour le parcours d'une liste chaînée

Parcours d'une structure de données : itérateurs

Pour parcourir les éléments d'une collection on utilise un **itérateur**.
L'API Java définit une interface `java.util.Iterator<E>` (extraits) :

`boolean hasNext()` Returns true if the iteration has more elements.

`E next()` Returns the next element in the iteration.

`void remove()` Removes from the underlying collection the last element returned by the iterator (optional operation)

`ListIterator<E>` parcours avant/arrière (previous(), hasPrevious()) + add(E e), set(E e)

Parcours d'une structure de données : itérateurs

Usage

Exemple du parcours d'une liste en Java

```
Collection<Recyclable> trashcan = new ArrayList<Recyclable>();
trashcan.add(new Paper()); // upcast vers Recyclable
trashcan.add(new Battery()); // implicite

// iterateur sur la collection
Iterator<Recyclable> it = trashcan.iterator();
while(it.hasNext()){
    Recyclable ref = it.next(); // it.next() du type Recyclable
    ref.recycle();
}
```


Parcours d'une structure de données : itérateurs

Attention !

- **Il ne faut pas parcourir une liste en utilisant** `get(int idx)`
- **Solution :** Itérateur

Pourquoi ne faut-il pas écrire :

```
List<...> l = ...;
for(int i = 0; i < l.size(); i ++){
    //utilisation de l.get(i)
}
```

```
.../java/test$ java TestCollection 20000
*** parcours LinkedList avec iterateur
pacours 20000 elements : 7 ms
*** parcours LinkedList avec get(i)
parcours 20000 elements : 480 ms
```

Parcours d'une structure de données : itérateurs

Remarque

- Possibilité d'utiliser **la syntaxe « à la for-each » pour itérer** sur les collections
- Cette syntaxe est possible sur les tableaux et toutes les classes qui implémentent l'interface `Iterable<T>`

```
Collection<Recyclable> trashcan = new ArrayList<Recyclable>();
trashcan.add(new Paper());
trashcan.add(new Battery());

for(Reclyclable r : trashcan){
    r.recycle();
}
```


Parcours d'une structure de données : itérateurs

Iterable

L'interface `java.lang.Iterable<T>` est définie par la méthode :

```
public Iterator<T> iterator();
```

Les objets des classes qui implémentent cette méthode pourront être utilisés dans une boucle for-each.

```
public class Agence implements Iterable<Voiture> f {
    private List<Voiture> lesVoitures;
    ...
    public Iterator<Voiture> iterator() {
        return this.lesVoitures.iterator();
    }
}

Agence agence = ...
for(Voiture v : agence){
    //utiliser v
}
```

Parcours d'une structure de données : itérateurs

Attention

- Les `Iterator` sont **fail-fast** : si, après que l'itérateur a été créé, la collection attachée est modifiée autrement que par les `add` et `remove` de l'itérateur alors il lance une `ConcurrentModificationException`

$\Longrightarrow$ rupture du contract de l'itérateur

$\Longrightarrow$ Echec rapide et propre plutot qu'inchohérence

```
List<Livre> l = ...;
for(int i = 0 ; i < 5; i++){
    l.add(new Livre(...));
}

Iterator<Livre> itLivre = l.iterator();
Livre l = itLivre.next(); // ok
l.add(new Livre(...)); // modification de la liste
// => corruption de l'iterateur
l = itLivre.next(); // -> ConcurrentModificationException levee
```


Plan

Set<E>

interface Set<E>

Un ensemble est une ...
2 classes :

- `HashSet<E>` : pour test appartenance rapide

  API Doc « This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets »

- `TreeSet<E>` trié à partir d'une structure d'arbre (`SortedSet` : `first(), last()`)

  API Doc « This implementation provides guaranteed $\log(n)$ time cost for the basic operations (`add, remove and contains`) »

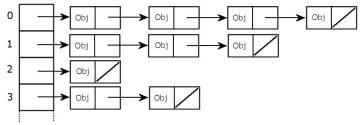
Remarque

⇒ Il est nécessaire d'implémenter `java.lang.Comparable / hashCode et equals`

Plan

Map<K,V>

Listes associatives (Clé, Valeur), dictionnaire, index, tables, etc.



Interface Map<K,V> : listes associatives clé -> valeur

- `HashMap<K,V>` : **table de hachage**, ajout et accès en temps constant

  API Doc « This implementation provides constant-time performance for the basic operations (`get and put`), assuming the hash function disperses the elements properly among the buckets »

- `TreeMap<K,V>` : en plus : clés triées

  API Doc « This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations »

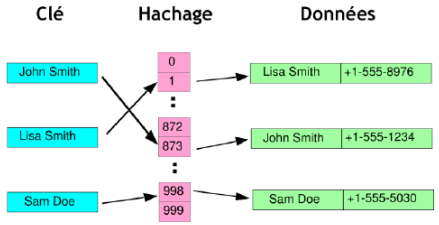
Remarque

Les `Map` ne sont pas des `Collections` $\Longrightarrow$ pas d'itérateur

Table de hachage

Définition

- Une **table de hachage** (*hash table* en anglais) est une structure de données permttant d'associer une valeur à une clé
- Il s'agit d'un tableau ne comportant **pas d'ordre** (un tableau est généralement indexé par des entiers)
- **L'accès à un élément** se fait en transformant la clé en une valeur de hachage par l'intermédiaire d'une fonction de hachage
- **Le hachage est un nombre qui permet la localisation des éléments dans le tableau**, *e.g.*, le hachage est l'indice de l'élément dans le tableau



Un annuaire représenté comme une table de hachage

Table de hachage

Opérations de base

Différentes opérations peuvent être effectuées sur une table de hachage :

- **création** d'une table de hachage
- **insertion** d'un nouveau couple (clé, valeur)
- **suppression** d'un élément
- **recherche de la valeur associée à une clé** (dans l'exemple de l'annuaire, retrouver le numéro de téléphone d'une personne)
- **destruction** d'une table de hachage (pour libérer la mémoire occupée)

Table de hachage

Méthodes de manipulation définit dans l'API Java pour l'interface `Map<K,V>` :

`V get(K key)` récupère la valeur associée à une clé

`void put(K key, V value)` ajoute un couple (clé, valeur)

`V remove(Object key)` supprime le couple associé à la clé key

`boolean containsKey(Object key)` teste l'existence d'une clé (equals)

`boolean containsValue(Object value)` teste l'existence d'une valeur (equals)

`Collection<V> values()` renvoie la collection des valeurs

`Set<K> keySet()` renvoie l'ensemble des clés

`Set<Map.Entry<K,V>> entrySet()` renvoie l'ensemble des couples (clé,valeurs) (objets `Map.Entry<K,V>`)

Usage

```
// associe un Auteur a un Livre
Map <Auteur,Livre> table = new HashMap <Auteur,Livre>();
Auteur auteur = new Auteur("Tolkien");

Livre livre1 = new Livre("Le Seigneur des Anneaux");
table.containsKey(auteur) // vaut false
table.put(auteur,livre1);
System.println(table.get(auteur).getTitre());
// -> affiche le Seigneur des Anneaux
table.containsKey(auteur) // vaut true
table.containsValue(livre1) // vaut true

Livre livre2 = new Livre("Le Silmarillion");
table.put(auteur,livre2);
System.println(table.get(auteur).getTitre());
// -> affiche le Silmarillion
table.containsValue(livre1) // vaut false
```


Parcours d'une Map

Attention

Pas d'itérateur « direct »

```
// associe Auteur (cle)  a Livre (valeur)
Map<Auteur,Livre> table = ...;
...
public void afficheMap() {
    Set<Auteur> lesCles = this.table.keySet();
    Iterator<Auteur> it cle = lesCles.iterator();
    //On itere sur les cles
    while (it cle.hasNext()) f
        Auteur a = it.next();
        System.print(a+" a ecrit "+ this.table.get(a));
    }
}

public void afficheMap() {
    for(Auteur a : this.table.keySet()) f
        System.print(a+" a ecrit "+ this.table.get(a));
    }
}
```

Parcours d'une Map

Attention

En manipulant les couples (`Map.entry`) :

```
public void afficheMap() {
    Set<Map.Entry<Auteur,Livre>> lesEntries = table.entrySet();
    Iterator<Map.Entry<Auteur,Livre>> it entry =
                            lesEntries.iterator();
    //On itere sur les couples (cle,valeur)
    while (it entry.hasNext()) {
        Map.Entry<Auteur,Livre> e = it entry.next();
        System.print(e.getKey()+" a ecrit "+ e.getValue());
    }
}

public void afficheMap(){
    for(Map.Entry<Auteur,Livre> entry : table.entrySet()){
        System.print(entry.getKey()+" ecrit "+ entry.getValue());
    }
}
```


Complexité : table de hachage

Avantages

- Tout comme les tableaux, **les tables de hachage permettent un accès** .theta(1) en moyenne , quel que soit le nombre d'éléments dans la table
- Toutefois, le temps d'accès dans le pire des cas . .Theta(n)
- Comparées aux autres tableaux associatifs, **les tables de hachage sont surtout utiles lorsque le nombre d'entrées** .le.plus important

Inconvénients

- La position des éléments dans une table de hachage est aléatoire : cette structure n'est donc **pas adaptée pour accéder à des données triées**
- Cette structure n'est pas non plus adaptée au feuilletage (browsing) de données voisines
- Des types de structures de données comme les arbres équilibrés, généralement plus lents (en $\Theta(\log n)$) et un peu plus complexes à implémenter, maintiennent une structure ordonnée

Table de hachage

Choix d'une bonne fonction de hachage

- Le fait de créer un hachage à partir d'une clé peut engendrer un **problème de collision** : **à partir de deux clés différentes, la fonction de hachage peut renvoyer la même valeur de hachage**, et donc par conséquent donner accès à la même position dans le tableau
- Pour minimiser les risques de collisions, il faut donc **choisir soigneusement sa fonction de hachage**
- Les collisions étant en général résolues par des méthodes de recherche linéaire, **une mauvaise fonction de hachage (produisant beaucoup de collisions) va fortement dégrader la rapidité de la recherche**
- Un bon compromis est à trouver entre :
  - rapidité de la fonction hachage
  - taille à réserver pour l'espace de hachage
  - réduction du risque des collisions

Table de hachage

Calcul de hachage

Le calcul du hachage se fait généralement en deux temps :

1. Une fonction de hachage est utilisé pour obtenir un nombre entier à partir de la donnée d'origine
2. Ce nombre entier est converti en position dans la table en général en calculant le reste modulo la taille de la table.

Remarques

- Les tailles des tables de hachage sont souvent des nombres premiers, afin d'éviter les problèmes de diviseurs communs, qui créeraient un nombre important de collisions
- Une alternative est d'utiliser une puissance de deux, ce qui permet de réaliser l'opération modulo par de simples décalages, et donc de gagner en rapidité

Table de hachage

Problème

- **Un problème fréquent est le phénomène de *clustering*** qui désigne le fait que des valeurs de hachage se retrouvent côte à côte dans la table, formant des « clusters » (« grappes »). Ceci est pénalisant pour les techniques de résolution des collisions par adressage ouvert
- Les fonctions de hachage réalisant **une distribution uniforme des hachages sont donc les meilleures**, mais sont en pratique difficile à trouver

La fonction de hash primaire (la flèche en haut) calcule une adresse et génère une collision. La première case libre en ordre croissant, ici en bleu, est trouvée et utilisée, consolidant ainsi deux clusters, provoquant une congestion supplémentaire.

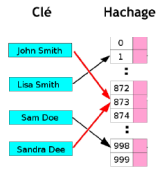
Table de hachage

Résolution des collisions

Lorsque deux clés ont la même valeur de hachage, ces clés ne peuvent être stockées à la même position, on doit alors employer une ... stratégie de résolution des collisions

Idée sous-jacente

- Dans le cas le plus favorable où la fonction de hachage a une distribution uniforme, **il y a 95% de chances d'avoir une collision dans une table de taille 1 million avant qu'elle ne contienne 2500 éléments**
- De nombreuses stratégies de résolution des collisions existent mais les plus connues et utilisées sont le chaînage et l'adressage ouvert

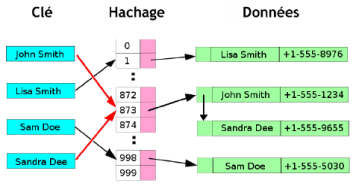


Exemple d'une collision (en rouge)

Résolution des collisions

Résolution par chaînage

- Chaque case de la table est en fait .une liste chaînée des dés qui ont le même hachage
- Une fois la case trouvée, **la recherche est alors linéaire** en la taille de la liste



Résolution des collisions par chaînage

Complexité

- **Dans le pire des cas où la fonction de hachage renvoie toujours la même valeur de hachage** quelle que soit la clé, la table de hachage devient alors une liste chaînée, et le temps de recherche est en . . .
- **L'avantage du chaînage est que la suppression d'une clé est facile ainsi que la recherche** (d'autres structures de données que les listes chaînées peuvent être utilisées. En utilisant un arbre équilibré, le coût théorique de recherche dans le pire des cas est en $\mathcal{O}(\log n)$)

Résolution des collisions

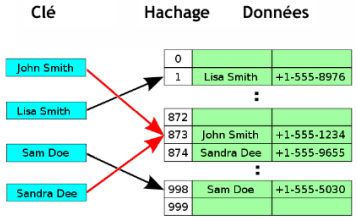
Résolution par adressage ouvert

- **L'adressage ouvert consiste, dans le cas d'une collision, à stocker les valeurs de hachage dans d'autres alvéoles vides**
- On appelle ce procédé une méthode de sondage : on essaie les alvéoles $h(k,0), h(k,1), \cdots, h(k,n)$ jusqu'à ce qu'on trouve une alvéole vide
- Il existe 3 méthodes de sondage

Sondage linéaire

Soit $H : U \to 0, \cdots, N-1$ une fonction de hachage. La fonction de sondage linéaire sera :

$$h(k,i) = (H(k) + i) \mod N, \quad \text{avec} \quad i = 0, 1, \cdots, N-1$$



Résolution des collisions par adressage ouvert et sondage linéaire

Résolution des collisions

Résolution par adressage ouvert

- **L'adressage ouvert consiste, dans le cas d'une collision, à stocker les valeurs de hachage dans d'autres alvéoles vides**
- On appelle ce procédé une méthode de sondage : on essaie les alvéoles $h(k,0), h(k,1), \cdots, h(k,n)$ jusqu'à ce qu'on trouve une alvéole vide
- Il existe 3 méthodes de sondage

Sondage quadratique

Soit $H : U \to 0, \cdots, N-1$ une fonction de hachage. La fonction de sondage quadratique sera :

$$h(k,i) = (H(k) + c_1.i + c_2.i^2) \mod N, \quad \text{avec} \quad i = 0, 1, \cdots, N-1, c_1 \quad \text{et} \quad c_2 \neq 0$$

Double hachage

Soient H_1 et $H_2 : U \to 0, \cdots, N-1$ deux fonctions de hachage. La fonction de sondage par double hachage sera :

$$h(k,i) = (H_1(k) + i.H_2(k)) \mod N, \quad \text{avec} \quad i = 0, 1, \cdots, N-1, c_1 \quad \text{et} \quad c_2 \neq 0$$

Résolution des collisions

Résolution par adressage ouvert

- **L'adressage ouvert consiste, dans le cas d'une collision, à stocker les valeurs de hachage dans d'autres alvéoles vides**
- On appelle ce procédé une méthode de sondage : on essaie les alvéoles $h(k,0), h(k,1), \cdots, h(k,n)$ jusqu'à ce qu'on trouve une alvéole vide
- Il existe 3 méthodes de sondage

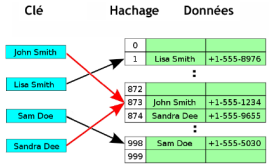
Comparaisons

- **Le sondage linéaire possède la meilleure performance en termes de cache**, mais est sensible à l'effet de clustering
- Le double hachage ne permet pas d'utiliser le cache efficacement, mais permet de réduire le clustering, au prix d'une complexité plus élevée
- **Le sondage quadratique se situe ...**

Opérations de base

Recherche Lors d'une recherche, si l'alvéole obtenue par hachage direct ne permet pas d'obtenir la bonne clé, une recherche sur les cases obtenues par une méthode de sondage est effectuée jusqu'à trouver la clé, ou tomber sur une alvéole vide, ce qui indique qu'aucune clé de ce type n'appartient à la table

Suppression Dans un adressage ouvert, la suppression d'un élément de la table de hachage est délicate. Dans le schéma ci-après, si l'on supprime « John Smith » sans précaution, on ne retrouvera plus « Sandra Dee ». La manière la plus simple de s'en sortir est de ne pas vider l'alvéole où se trouvait « John Smith », mais d'y placer le mot « Supprimé ». On distinguera ainsi les alvéoles vides des alvéoles où un nom a été effacé : une alvéole contenant « Supprimé » sera considérée comme occupée lors d'une suppression, mais vide lors d'une insertion.



Performances

Facteur de charge

- Une indication critique des performances d'une table de hachage est ... le facteur de charge (proportion d'utilisation de la table)
- Plus le facteur de charge est proche de 100 %, plus le **nombre de sondages à effectuer devient important**
- Lorsque la table est presque pleine, les **algorithmes de sondage peuvent même échouer** : ils peuvent ne plus trouver d'alvéole vide, alors qu'il y en a encore de la place

⇒ Le facteur de charge est limié à 80%

⇒ Des facteurs de charge faibles ne sont pas pour autant significatifs de bonne performance, en particulier si la fonction de hachage est mauvaise et génère du clustering

Plan

Références

Bibliographie

Des éléments de ce cours sont empruntés de
[Müller(2014), Nicod(2007), Knuth(1997), Routier(2012)]


D. Knuth.
The art of computer programming, vol. 3 : Sorting and searching.
Addison-Wesley, 1997.


D. Müller.
Structures de données avancées.
Apprendre en ligne, 2014.
URL http://www.apprendre-en-ligne.net/info/structures/index.html.


J. M. Nicod.
Les tables de hachage.
UFR des Sciences et Techniques, Université de Franche-comté, 2007.
URL http://lifc.univ-fcomte.fr/~nicod/slidesHashTableL3.pdf.


J. C. Routier.
CM 2 – Collections et tables de hachage.
Module Programmation Orientée Objet, Licence mention Informatique,
Université Lille 1, 2012.